Assignment Title: Time Complexity Analysis of Algorithms

Objective: Determine the time complexity of algorithms by analyzing elementary operations and applying Big-O rules.

1. **Theoretical Analysis** – Determining time complexity using **Big-O notation** by analyzing elementary operations.

2. **Empirical Analysis** – Running algorithms for different input sizes and measuring execution time.

Section 1: Theoretical Analysis of Time Complexity using Big-O Notation

Analyze the following algorithms and determine their **Big-O time complexity**. Justify your answers by:

1. Identifying loops, recursion, and key operations.

2. Counting iterations in relation to input size `n`.

3. Applying rules (drop constants, keep dominant terms).

Task 1: Searching Algorithms Complexity Analysis

Case 1: Jump Search

```
def jump_search(arr, x):
    import math
    n = len(arr)
    step = int(math.sqrt(n))
    prev = 0
    while arr[min(step, n)-1] < x:
        prev = step
        step += int(math.sqrt(n))
        if prev >= n:
            return -1
    for i in range(prev, min(step, n)):
        if arr[i] == x:
            return i
    return -1
```

- **Determine the time complexity of Jump Search and compare with Binary Search.**

ase 2: Interpolation Search

```
def interpolation_search(arr, x):
    low, high = 0, len(arr) - 1
    while low <= high and x >= arr[low] and x <= arr[high]:
        pos = low + ((x - arr[low]) * (high - low) // (arr[high] - arr[low]))
        if arr[pos] == x:
            return pos
        elif arr[pos] < x:
            low = pos + 1
        else:
            high = pos - 1
    return -1
```

- **Explain how Interpolation Search achieves O(log log n) in the best case scenario.**

Task 2: Sorting Algorithms Complexity Analysis

Case 1: Insertion Sort

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

- **Analyze why Insertion Sort performs efficiently for nearly sorted data.**

Case 2: Heap Sort

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

- **Compare Heap Sort with QuickSort in terms of stability and performance.**

Section 2: Empirical Analysis of Time Complexity

Task 1: Measuring Execution Time

1. Implement **Linear Search and Binary Search** and measure execution time for increasing input sizes.
2. Implement **Selection Sort and QuickSort**, then compare their execution times.
3. Define **worst-case input scenarios** for all algorithms.
4. **Measure execution time** for input sizes: `[1000, 5000, 10000, 50000, 100000]`

Example: Measuring Execution Time of Jump Search

```
import time

def measure_time_search(algorithm, arr, x):
    start_time = time.time()
    algorithm(arr, x)
    end_time = time.time()
    return end_time - start_time

sizes = [1000, 5000, 10000, 50000, 100000]
times = [measure_time_search(jump_search, list(range(n)), -1) for n in sizes]

for size, t in zip(sizes, times):
    print(f"Input Size: {size}, Execution Time: {t:.6f} seconds")
```

- **Interpret the results and verify complexity.**

Task 2: Plotting Execution Time vs. Input Size

1. Compare execution times of **Insertion Sort and Heap Sort**.

1. Visualize execution time trends by drawing a graph.
2. **Describe the trend** (linear, quadratic, logarithmic, etc.).
3. **Compare with theoretical complexity**.

Questions:

- Does the plot match the theoretical Big-O complexity? Why or why not?
- List **2 factors** that could cause discrepancies (e.g., hardware, programming language optimizations).

Task 3: Comparing Theoretical vs. Empirical Complexity

Fill in the table by running multiple algorithms and analyzing execution times.

Algorithm	Theoretical Complexity	Observed Trend
Jump Search	$O(\sqrt{n})$	Sub-linear Growth
Interpolation Search	$O(\log \log n)$	Log-logarithmic Growth
Insertion Sort	$O(n^2)$	Quadratic Growth
Heap Sort	$O(n \log n)$	n log n Growth

- **Discuss any deviations caused by hardware optimizations or caching effects.**

Submission Guidelines

Students must submit:

1. **Big-O Analysis Report** with explanations.
2. **Empirical Analysis Report** with graphs and execution time comparisons.
3. **Conclusion** summarizing observations and findings.